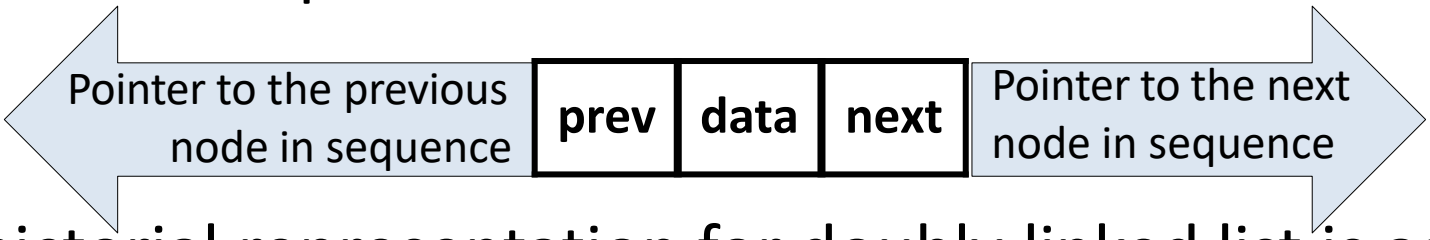
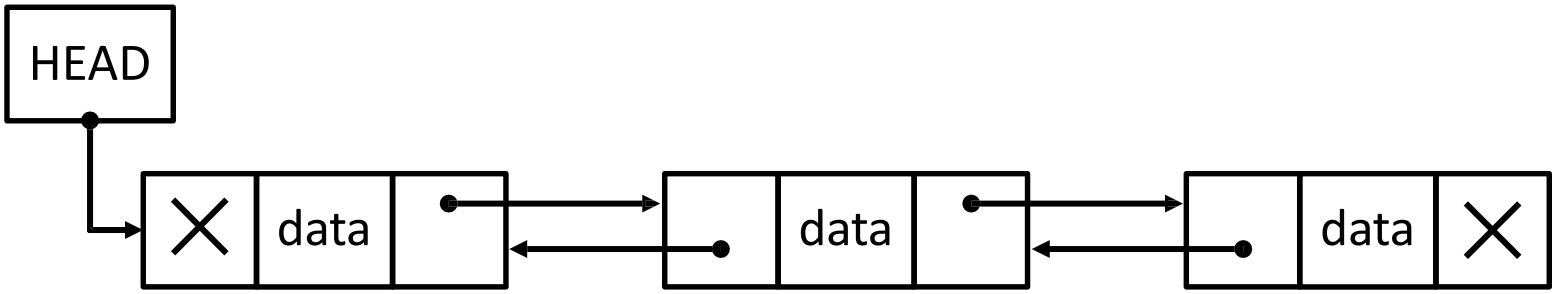
# Doubly Linked List

# Introduction

- Sequence of elements in which every element has links to its previous element and next element in the sequence.
- Each node contains three fields: data, link to the next node, and link to the previous node.




- The pictorial representation for doubly linked list is as shown below:



# Contd…

- Advantages:
  - Can be traversed in either direction.
  - Some operations, such as deletion and insertion before a node, become easier.

- Disadvantages:
  - Requires more space.
  - List manipulations are slower.
  - Greater chances of having bugs.

# Creation

```
struct node
{   int data;
    struct node *next, *prev; };
```

- Define node structure.
- Declare a NULL initialized head node pointer to create an empty list.

```
struct node *head = NULL;
```

- Dynamically allocate memory for a node and initialize all members of a node.

```
struct node *temp =
                    (struct node *) malloc (sizeof(struct node));
int num;
scanf("%d",&num);
temp -> data = num;
temp -> prev = temp -> next = NULL;
```

```
class Node {
public:
    int data;
    Node* next;
    Node* prev;

    Node(int num)
{   data= num,
     next= nullptr,
     prev= nullptr
}
};
class DoublyLinkedList {
public:
    Node* head;
    DoublyLinkedList()
   {
     head= nullptr}
```

```
int main() {
    DoublyLinkedList dll;
```

# Contd…

```
head = temp;
```

- Link the new node temp in the existing empty list.
- Again dynamically allocate memory for a node and initialize all members of a node.

```
*temp = (struct node *) malloc (sizeof(struct node));
scanf("%d",&num);
temp -> data = num;
temp -> prev = temp -> next = NULL;
```

- Link the new node temp in the existing list at head.

```
temp -> next = head; head -> prev = temp;
head = temp;
```

- This process is repeated for all the nodes. A node can inserted anywhere in the list.

# Search an element in the list

- **Algorithm** search(head, num)
- **Input**: Pointer to the first node (**head**) and a value to search (**num**).
- **Output**: Appropriate message will be displayed.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Return.
4. Initialize a node pointer (**temp**) with **head**.
5. while (**temp** is not **NULL** AND **temp**[**data**] is not equal to **value**)
6. **temp** = **temp**[**next**]
7. if (**temp** is **NULL**)
8. Print [**Element not found**].
9. Else
10. Print [**Element found**].

# Search an element in the list (same as SLL)

```
// Search for an element
  bool search(int value) {
     Node* current = head;
     while (current) {
        if (current->data == value) {
           return true;
        }
        current = current->next;
     }
     return false;
  }
```

# Display elements in the list

- **Algorithm** display(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: Display all the elements present in the list.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Return.
4. Initialize a node pointer (**temp**) with **head**.
5. while (**temp** is not **NULL**)
6. Print [**temp**[**data**]].
7. **temp** = **temp**[**next**].

# Insertion at beginning of the list

- **Algorithm** insertBeg(head, num)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output**: Node with value **num** gets inserted at the first position.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**.
3. **temp**[**prev**] = **NULL**.
4. **temp**[**next**] = **head**.
5. if (**head** == **NULL**)
6. **head** = **temp**.
7. else
8. **head**[**prev**] = **temp**.
9. **head** = **temp**.

# Insertion at beginning of the list

```
// Insert at the beginning
   void insertAtBegin(int num) {
      Node* newNode = new Node(num);
      if (!head) {
         head = newNode;
      } else {
         newNode->next = head;
         head->prev = newNode;
         head = newNode;
      }
   }
```

# Insertion at end of the list

- **Algorithm** insertEnd(head, num)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output**: Node with value **num** gets inserted at the last position.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**
3. **temp**[**prev**] = **temp**[**next**] = **NULL**
4. If (**head** == **NULL**)
5. **head** = **temp**
6. Else
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1**[**next**] is not equal to **NULL**)
9. **temp1** = **temp1**[**next**]
10. **temp1**[**next**] = **temp**
11. **temp**[**prev**] = **temp1**

# Insertion at end of the list

```
// Insert at the end
  void insertAtEnd(int num) {
     Node* newNode = new Node(num);
     if (!head) {
        head = newNode;
     } else {
        Node* current = head;
        while (current->next) {
           current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
     }
  }
```

# Insertion after a specific value in the list

- **Algorithm** insert(head, num, value)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**) after an existing **value**.
- **Output**: Node with value **num** gets inserted after node with **value**.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**
3. **temp**[**prev**] = **temp**[**next**] = **NULL**
4. If (**head** == **NULL**)
5. **head** = **temp**
6. Else
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1** is not **NULL** AND **temp1**[**data**] is not equal to **value**)
9. **temp1** = **temp1**[**next**]

# Contd…

```
10.     if (temp1 == NULL)
11.             print [Node is not present in the list]
12.     else if (temp1[next] is NULL)
13.             temp1[next] = temp
14.             temp[prev] = temp1
15.     else
16.             temp[prev] = temp1
17.             temp[next] = temp1[next]
18.             temp1[next] = temp
19.             (temp[next])[prev]= temp
20.     end if (line 10).
21.  End if (line 4).
```

# Insertion after a specific value in the list

```
// Insert after a specific value
   void insertAfterValue(int value, int num) {
      Node* newNode = new Node(num);
      Node* current = head;
      while (current) {
         if (current->data == value) {
            newNode->next = current->next;
            newNode->prev = current;
            if (current->next) {
               current->next->prev = newNode;
            }
            current->next = newNode;
            return;
         }
         current = current->next;
      }
      std::cout << "Value " << value << " not
found in the list." << std::endl;
   }
```

# Delete from beginning of the list

- **Algorithm** deleteBeg(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: The first node gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. **head = head**[**next**]
6. if (**head != NULL**)
7. **head**[**prev**] = **NULL**
8. else
9. **head** = **NULL**
10. Release the memory location pointed by **temp**.

# Delete from beginning of the list

```
// Delete at the beginning

   void deleteAtBegin() {
      if (!head) {
         std::cout << "List is Empty. Nothing to
delete." << std::endl;
         return;
      }
      Node* temp = head;
      head = head->next;
      if (head) {
         head->prev = nullptr;
      }
      delete temp;
   }
```

# Delete from end of the list

- **Algorithm** deleteEnd(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: The last node gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. while (**temp**[**next**] is not **NULL**)
6. **temp = temp[next]**
7. if (**temp** == **head**)
8. **head** = **NULL**
9. else
10. (**temp**[**prev**])[**next**] = **NULL**
11. Release the memory location pointed by **temp**.
12. end if

# Delete from end of the list

```
// Delete at the end
   void deleteAtEnd() {
      if (!head) {
         std::cout << "List is Empty. Nothing to
delete." << std::endl;
         return;
      }
      Node* current = head;
      while (current->next) {
         current = current->next;
      }
      if (current->prev) {
         current->prev->next = nullptr;
      } else {
         head = nullptr;
      }
      delete current;
   }
```

# Delete a specific node from the list

- **Algorithm** deleteSpecific(head,num)
- **Input**: Pointer to the first node (**head**) and a value **num** to be deleted.
- **Output**: The node with value **num** gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. while (**temp** != **NULL** AND **temp**[**data**] != **value**)
6. **temp = temp[next]**
7. if (**temp** is **NULL**)
8. Print [**Element not found**].
9. Return.

# Contd…

10. else if (**temp == head**)
11. deleteBeg(head)
12. else if (**temp**[**next**] **== NULL**)
13. deleteEnd(head)
14. else
15. (**temp**[**prev**])[**next**] = **temp**[**next**]
16. (**temp**[**next**])[**prev**] = **temp**[**prev**]
17. Release the memory location pointed by **temp**.
18. end if (line 7).
19. end if (line 1).

# Delete a specific node from the list

```
void deleteAfterValue(int value) {
    Node* current = head;
    while (current) {
        if (current->data == value) {
            Node* temp = current->next;
            if (temp) {
                current->next = temp->next;
                if (temp->next) {
                    temp->next->prev = current;
                }
                delete temp;
            } else {
                std::cout << "No element to delete after " << value << "."
<< std::endl;
            }
            return;
        }
        current = current->next;
    }
    std::cout << "Value " << value << " not found in the list." <<
std::endl;
}
```

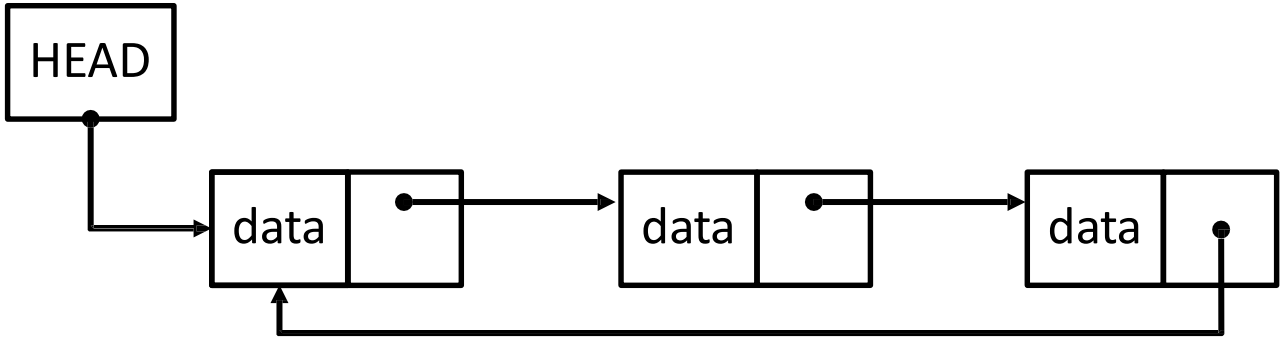
# Circular Linked List

# Introduction

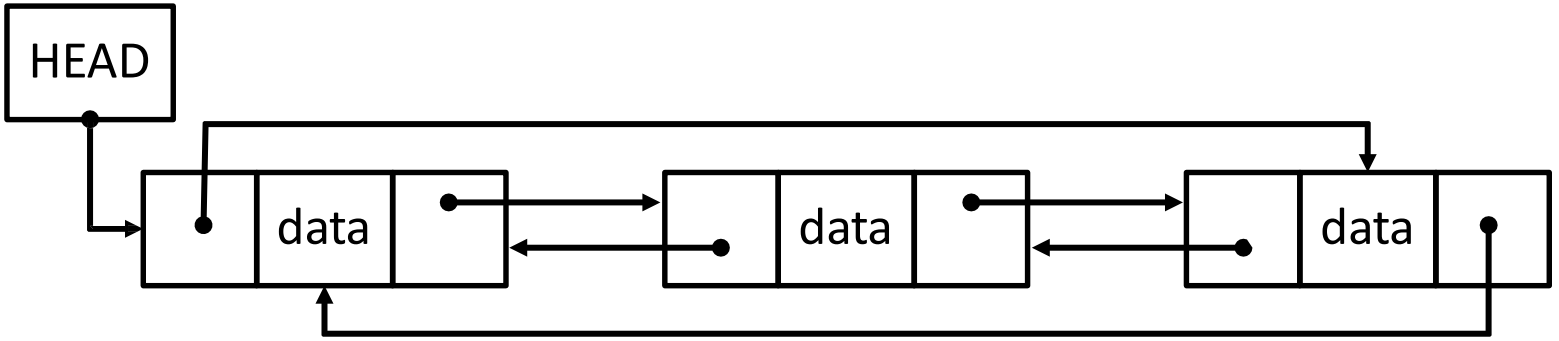
- The first element points to the last element and the last element points to the first element.
- There is no NULL node.
- While traversal, get back to a node from where you have started.
- Pointer to any node can serve as a handle to the complete list.
- Both singly and doubly linked lists can be circular.

# Contd…

- Singly linked list as circular




- Doubly linked list as circular



# Creation

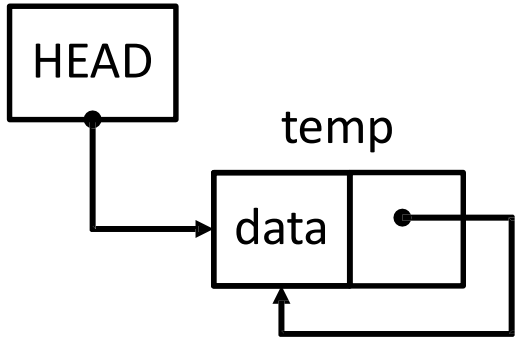
- Define node structure.
- Declare a NULL initialized head node pointer to create an empty list.
- Dynamically allocate memory for a node and initialize all members of a node.
- Link the new node temp in the existing empty list.
- Again dynamically allocate memory for a node and initialize all members of a node.
- Link the new node temp in the existing list at head.
- This process is repeated for all the nodes. A node can be inserted anywhere in the list.

# Contd…

- Link the new node temp in the existing empty list.

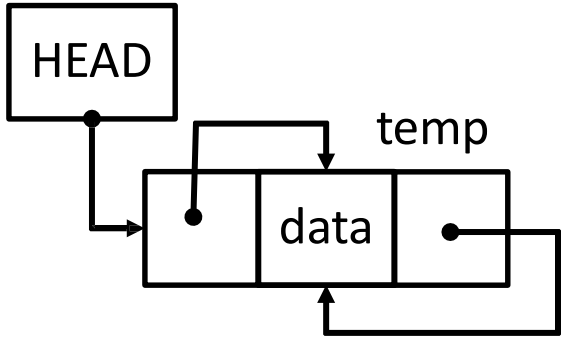
- Singly

```
head = temp;
temp -> next = head;
```




- Doubly

```
head = temp;
temp -> prev = head;
temp -> next = head;
```



# Contd…

- Link the new node temp in the existing list at head.

- Singly: temp1 is a node pointer pointing to the last node in a linked list.

```
temp -> next = head;
temp1 -> next = temp;
head = temp;
```

- Doubly

```
temp -> next = head;
temp -> prev = head -> prev;
head -> prev = temp;
temp -> prev -> next = temp;
head = temp;
```

# Search an element in the list

- **Algorithm** search(head, num)
- **Input**: Pointer to the first node (**head**) and a value to search (**num**).
- **Output**: Appropriate message will be displayed.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Return.
4. Initialize a node pointer (**temp**) with **head**.
5. while (**temp**[**next**] != **head** AND **temp**[**data**] != **value**)
6. **temp** = **temp**[**next**]
7. if (**temp**[**data**] == **value**)
8. Print [**Element found**].
9. Else
10. Print [**Element not found**].

# Search an element in the list

```
bool search(int value) {
        if (!head) {
            std::cout << "List is Empty" << std::endl;
            return false;
        }

        Node* temp = head;
        do {
            if (temp->data == value) {
                std::cout << "Element found" <<
std::endl;
                return true;
            }
            temp = temp->next;
        } while (temp != head);

        std::cout << "Element not found" <<
std::endl;
        return false;
    }
```

# Display elements in the list

- **Algorithm** display(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: Display all the elements present in the list.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Return.
4. Initialize a node pointer (**temp**) with **head**.
5. while (**temp**[**next**] is not **head**)
6. Print [**temp**[**data**]].
7. **temp** = **temp**[**next**].
8. Print [**temp**[**data**]].

# Display elements in the list

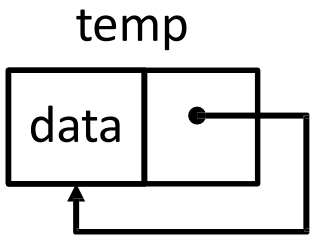
```
void display() {
    if (!head) {
        std::cout << "List is Empty" << std::endl;
        return;
    }

    Node* temp = head;
    do {
        std::cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    std::cout << std::endl;
  }
};
```

# Insertion at beginning of the list (singly)

- **Algorithm** insertBeg(head, num)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output**: Node with value **num** gets inserted at the first position.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**.
3. if (**head** == **NULL**)
4. **temp**[**next**] = **temp**.



# Contd…

5. else
6. **temp**[**next**] = **head**.
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1**[**next**] is not equal to **head**)
9. **temp1** = **temp1**[**next**]
10. **temp1**[**next**] = **temp**.
11. end if (line 3).
12. **head** = **temp**.

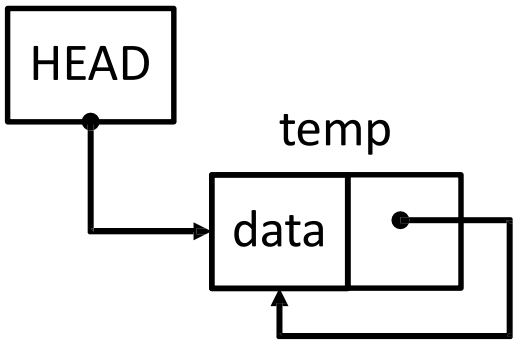
# Insertion at beginning of the list (singly)

```
void insertAtBegin(int num) {
        Node* newNode = new Node(num);
        if (!head) {
            head = newNode;
            newNode->next = head;
        } else {
            Node* current = head;
            while (current->next != head) {
                current = current->next;
            }
            current->next = newNode;
            newNode->next = head;
            head = newNode;
        }
    }
```

# Insertion at end of the list (singly)

- **Algorithm** insertEnd(head, num)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output**: Node with value **num** gets inserted at the last position.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**
3. If (**head** == **NULL**)
4. **temp**[**next**] = **temp**
5. **head** = **temp**
6. Else
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1**[**next**] is not equal to **head**)
9. **temp1** = **temp1**[**next**]
10. **temp1**[**next**] = **temp**
11. **temp**[**next**] = **head**



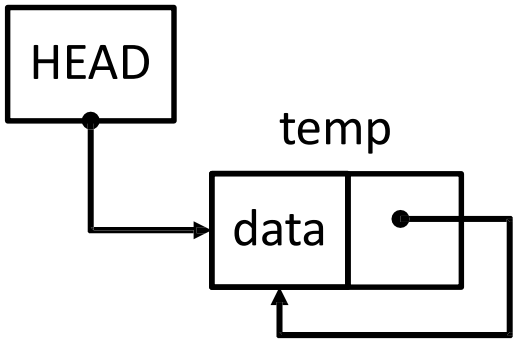
# Insertion at end of the list (singly)

```
void insertAtEnd(int num) {
      Node* newNode = new Node(num);
      if (!head) {
         head = newNode;
         newNode->next = head;
      } else {
         Node* current = head;
         while (current->next != head) {
            current = current->next;
         }
         current->next = newNode;
         newNode->next = head;
      }
   }
```

# Insertion after a specific value in the list (singly)

- **Algorithm** insert(head, num, value)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**) after an existing **value**.
- **Output**: Node with value **num** gets inserted after node with **value**.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**
3. If (**head** == **NULL**)
4. **temp**[**next**] = **temp**
5. **head** = **temp**



# Contd…

6. else
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1**[**next**] != **head** AND **temp1**[**data**] != **value**)
9. **temp1** = **temp1**[**next**]
10. if (**temp1**[**next**] == **head** AND **temp1**[**data**] != **value**)
11. print [**Node is not present in the list**]
12. else
13. **temp**[**next**] = **temp1**[**next**]
14. **temp1**[**next**] = **temp**
15. end if (line 10).
16. End if (line 3).

# Delete from beginning of the list (singly)

- **Algorithm** deleteBeg(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: The first node gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize node pointers (**temp** and **temp1**) with **head**.
5. while (**temp1**[**next**] is not equal to **head**)
6. **temp1** = **temp1**[**next**]
7. if (**temp1** == **head**)
8. **head** == **NULL**
9. else
10. **temp1**[**next**] = **head**[**next**].
11. **head = head**[**next**]
12. Release the memory location pointed by **temp**.

# Delete from beginning of the list (singly

```
// Delete at the beginning
   void deleteAtBegin() {
      if (!head) {
         std::cout << "List is Empty. Nothing to
delete." << std::endl;
         return;
      }
      Node* temp = head;
      if (head->next == head) {
         head = nullptr;
      } else {
         Node* current = head;
         while (current->next != head) {
            current = current->next;
         }
         head = head->next;
         current->next = head;
      }
      delete temp;
   }
```

# Delete from end of the list (singly)

- **Algorithm** deleteEnd(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: The last node gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. while (**temp**[**next**] is not **head**)
6. initialize a node pointer (**pre**) with **temp**.
7. **temp** = **temp[next]**
8. if (**temp** == **head**)
9. **head** = **NULL**
10. else
11. **pre**[**next**] = **head**
12. Release the memory location pointed by **temp**.

# Delete from end of the list (singly)

```
// Delete at the end
   void deleteAtEnd() {
      if (!head) {
         std::cout << "List is Empty. Nothing to
delete." << std::endl;
         return;
      }
      Node* temp = head;
      if (head->next == head) {
         head = nullptr;
      } else {
         Node* current = head;
         Node* prev = nullptr;
         while (current->next != head) {
            prev = current;
            current = current->next;
         }
         prev->next = head;
      }
      delete temp;
   }
```

# Delete a specific node from the list (singly)

- **Algorithm** deleteSpecific(head,num)
- **Input**: Pointer to the first node (**head**) and a value **num** to be deleted.
- **Output**: The node with value **num** gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. while (**temp**[**next**] != **head** AND **temp**[**data**] != **value**)
6. initialize a node pointer (**pre**) with **temp**.
7. **temp = temp[next]**
8. if (**temp**[**data**] != **value**)
9. Print [**Element not found**].
10. Return.

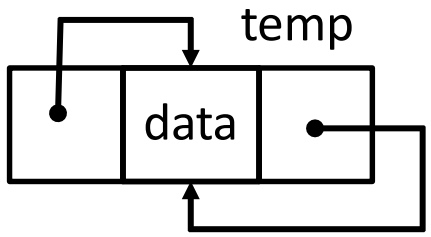
# Contd…

11. else if (**temp == head**)
12. deleteBeg(**head**)
13. else if (**temp**[**next**] == **head**)
14. deleteEnd(**head**)
15. else
16. **pre**[**next**] = **temp**[**next**]
17. Release the memory location pointed by **temp**.
18. end if (line 8).
19. end if (line 1).

# Insertion at beginning of the list (doubly)

- **Algorithm** insertBeg(head, num)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output**: Node with value **num** gets inserted at the first position.

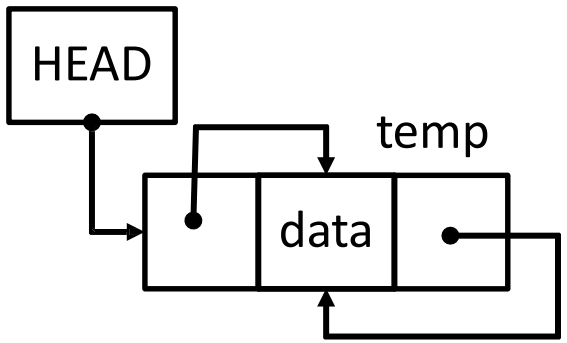
1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**.
3. if (**head** == **NULL**)
4. **temp**[**next**] = **temp**[**prev**] = **temp**.
5. else
6. **temp**[**next**] = **head**.
7. **temp**[**prev**] = **head**[**prev**].
8. **head**[**prev**] = **temp**.
9. (**temp**[**prev**])[**next**] = **temp**.
10. **head** = **temp**.



# Insertion at end of the list (doubly)

- **Algorithm** insertEnd(head, num)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output**: Node with value **num** gets inserted at the last position.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**
3. If (**head** == **NULL**)
4. **temp**[**prev**] = **temp**
5. **temp**[**next**] = **temp**
6. **head** = **temp**



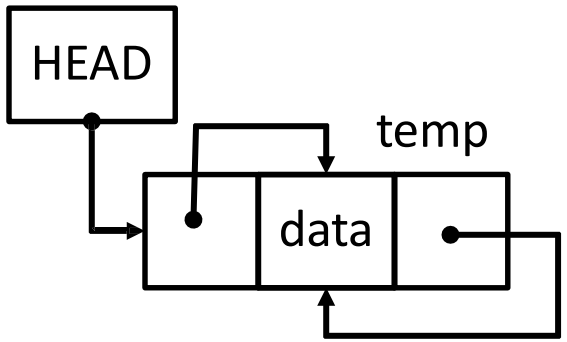
# Contd…

7. else
8. **temp**[**next**] = **head**
9. **temp**[**prev**] = **head**[**prev**]
10. (**head**[**prev**])[**next**] = **temp**
11. **head**[**prev**] = **temp**

# Insertion after a specific value in the list (doubly)

- **Algorithm** insert(head, num, value)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**) after an existing **value**.
- **Output**: Node with value **num** gets inserted after node with **value**.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**
3. If (**head** == **NULL**)
4. **temp**[**prev**] = **temp**[**next**] = **temp**
5. **head** = **temp**
6. Else
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1**[**next**] != **head** AND **temp1**[**data**] != **value**)
9. **temp1** = **temp1**[**next**]



# Contd…

10. if (**temp1**[**next**] == **head** AND **temp1**[**data**] != **value**)
11. print [**Node is not present in the list**]
12. else
13. **temp**[**prev**] = **temp1**
14. **temp**[**next**] = **temp1**[**next**]
15. **temp1**[**next**] = **temp**
16. (**temp**[**next**])[**prev**] = **temp**
17. end if (line 10).
18. End if (line 3).

# Delete from beginning of the list (doubly)

- **Algorithm** deleteBeg(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: The first node gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. if (**temp**[**next**] == **head**)
6. **head** == **NULL**
7. else
8. (**head**[**prev**])[**next**] = **head**[**next**]
9. (**head**[**next**])[**prev**] = **head**[**prev**]
10. **head = head**[**next**]
11. Release the memory location pointed by **temp**.

# Delete from end of the list (doubly)

- **Algorithm** deleteEnd(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: The last node gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. while (**temp**[**next**] is not **head**)
6. **temp = temp[next]**
7. if (**temp** == **head**)
8. **head** = **NULL**
9. else
10. (**temp**[**prev**])[**next**] = **head**
11. **head**[**prev**] = **temp**[**prev**]
12. Release the memory location pointed by **temp**.
13. end if

# Delete a specific node from the list (doubly)

- **Algorithm** deleteSpecific(head,num)
- **Input**: Pointer to the first node (**head**) and a value **num** to be deleted.
- **Output**: The node with value **num** gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. while (**temp**[**next**] != **head** AND **temp**[**data**] != **value**)
6. **temp = temp[next]**
7. if (**temp**[**data**] != **value**)
8. Print [**Element not found**].
9. Return.

# Contd…

10. else if (**temp** == **head**)
11. deleteBeg(**head**)
12. else if (**temp**[**next**] == **head**)
13. deleteEnd(**head**)
14. else
15. (**temp**[**prev**])[**next**] = **temp**[**next**]
16. (**temp**[**next**])[**prev**] = **temp**[**prev**]
17. Release the memory location pointed by **temp**.
18. end if (line 7).
19. end if (line 1).